

Fig. 1

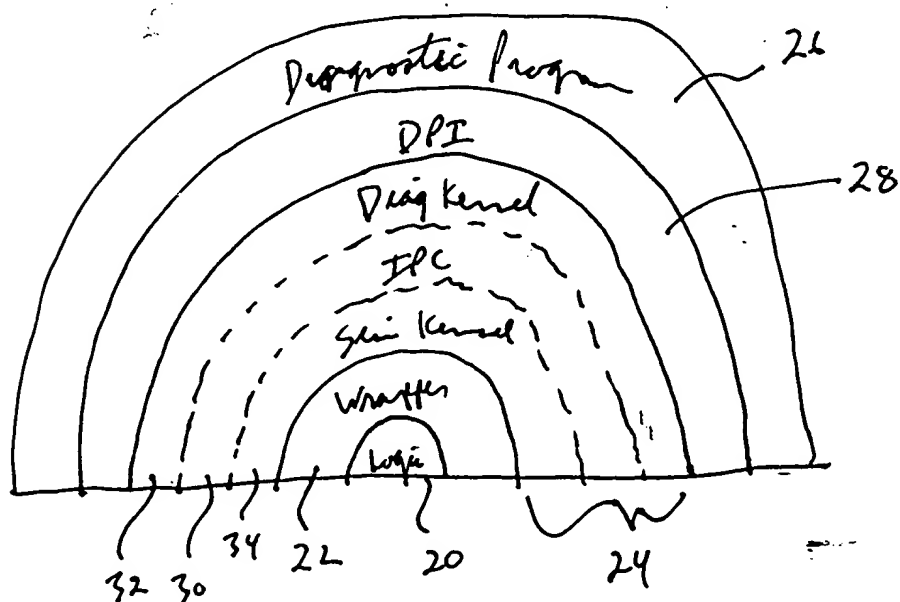
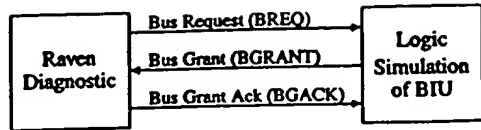


Fig. 2



40

Fig. 3

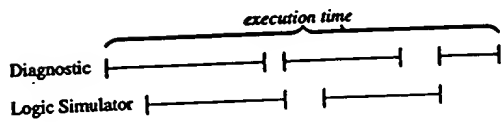


Fig. 4

TIME (s)

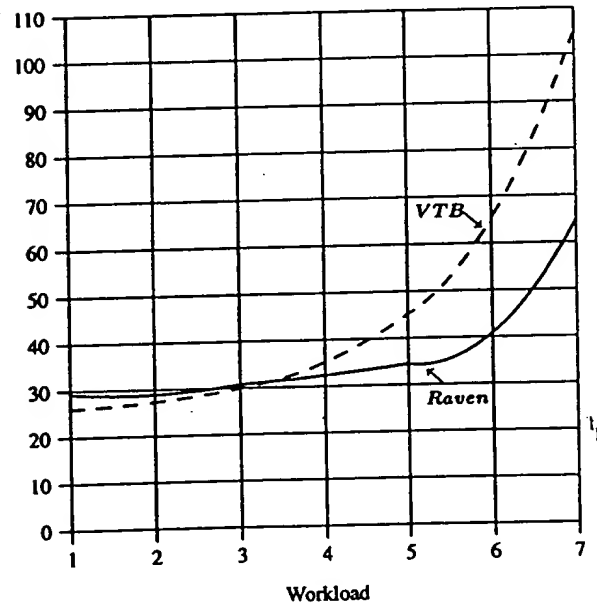


Fig. 5

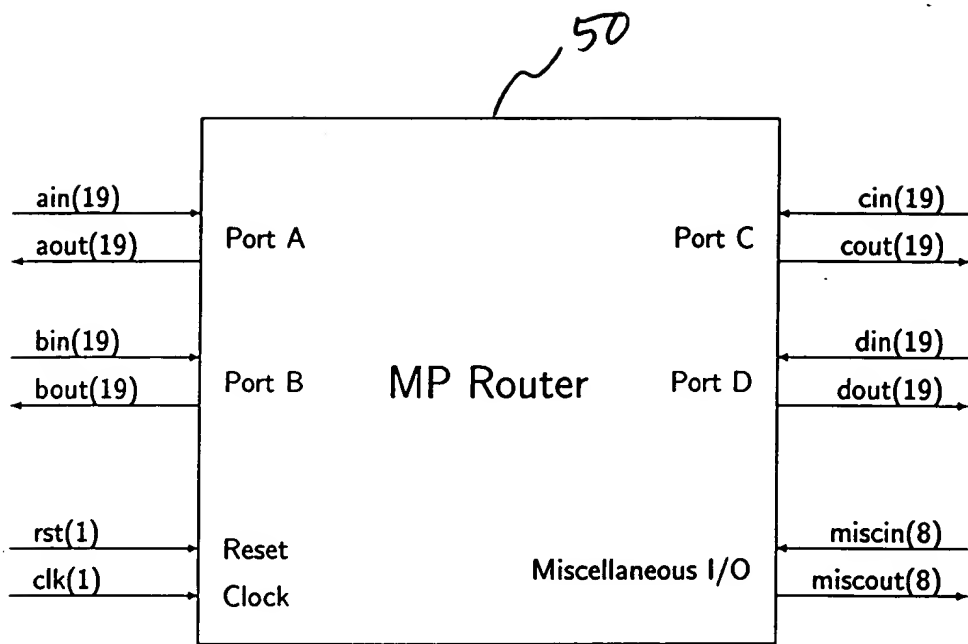


Fig. 6

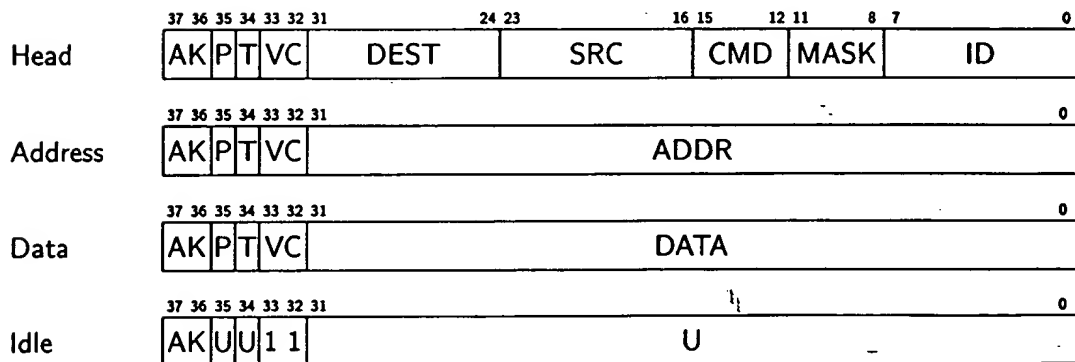


Fig. 7

|                |               |         |      |
|----------------|---------------|---------|------|
| Get Request    | Head, CMD=0x0 | Address |      |
| Put Request    | Head, CMD=0x1 | Address | Data |
| Get Response   | Head, CMD=0x0 | Data    |      |
| Put Response   | Head, CMD=0x1 |         |      |
| Error Response | Head, CMD=0xF |         |      |

Fig. 8

```

'timescale 1ns/100ps
module rtr_wrapper;

wire [19:0] Astmc, Artmc, Bstmc, Brtmc;
wire [19:0] Cstmc, Crtmc, Dstmc, Drtmc;
wire [7:0] miscin, miscout;
wire Astall, Bstall, Cstall, Dstall;
wire rst, clk;

initial begin
    clk = 0;
    forever #5 clk = ~clk;
end

rtr_top
    top(Astmc, Artmc, Bstmc, Brtmc, Cstmc, Crtmc, Dstmc, Drtmc,
        miscin, miscout, rst, clk);

rtr_stub Atmc_stub(0, Astmc, Artmc, Astall, rst, clk);
rtr_stub Btmc_stub(2, Bstmc, Brtmc, Bstall, rst, clk);
rtr_stub Ctmc_stub(4, Cstmc, Crtmc, Cstall, rst, clk);
rtr_stub Dtmc_stub(6, Dstmc, Drtmc, Dstall, rst, clk);

raven_apply_change #(1) Astall_stub(8, clk, Astall);
raven_apply_change #(1) Bstall_stub(9, clk, Bstall);
raven_apply_change #(1) Cstall_stub(10, clk, Cstall);
raven_apply_change #(1) Dstall_stub(11, clk, Dstall);

raven_apply_change #(1) miscin_stub(12, clk, miscin);
raven_verify_change #(1) miscout_stub(13, clk, miscout);
raven_apply_change #(1) rst_stub(14, clk, rst);

endmodule

```

Fig. 9

```
module rtr_stub(port, stmc, rtmc, stall, rst, clk)

input [31:0] port;
output [19:0] stmc;
input [19:0] rtmc;
input stall, rst, clk;
wire sak, rak;

rtr_xmtr xmtr(port, stmc, sak, rak, rst, clk);
rtr_rcvr rcvr(port + 1, rtmc, sak, stall, rst, clk);

endmodule
```

Fig. 10

```

module rtr_xmtr(port, stmc, sak, rak, rst, clk)

input [31:0] port;
output [19:0] stmc;
input [1:0] sak, rak;
input rst, clk;
wire [31:0] data;
wire [1:0] vc;
wire p, t, valid;
wire [3:0] block;

raven_apply_channel #(33,2)
  raven(port, clk, {p, data}, vc, t, valid, block, 1);

// Decode virtual channels
wire go0 = valid & (vc == 0);
wire go1 = valid & (vc == 1);
wire go2 = valid & (vc == 2);
wire go = go0 | go1 | go2;

// Encode the transmitted flit. Ensure idle output while reset
// is in progress (rak is already reset clean from receiver).
reg [37:0] out;
always @(rst or go or rak or p or t or vc or data)
  if (!rst && go)
    out = {rak, p, t, vc, data};
  else
    out = {rak, 1'b0, 1'b0, 2'h3, 32'h00000000};

// Disassemble the flit based on the phase of the clock.
stmc = clk ? out[37:19] : out[18:0];

// Decode remote acknowledges.
wire ak0 = (sak == 0);
wire ak1 = (sak == 1);
wire ak2 = (sak == 2);

// Track the free buffer space at the remote receiver.
rtr_free free0(block0, go0, ak0, rst, clk);
rtr_free free1(block1, go1, ak1, rst, clk);
rtr_free free2(block2, go2, ak2, rst, clk);

// Assemble the virtual channel blocks.
assign block = {1'b1, block2, block1, block0};

endmodule

```

Fig. 11

```

module rtr_free(block, go, ak, rst, clk)

output block;
input go, ak, rst, clk;
reg[3:0] free;

// Initialize free count at reset.
// Decrement free count when transmitting and increment when
// being acknowledged. If both happen simultaneously, net
// result is to do nothing.
always @(posedge clk)
    if (rst)
        free <= 8;
    else if (go && !ak)
        free <= free - 1;
    else if (!go && ak)
        free <= free + 1;

// Block when no space left.
assign block = (free == 0);

endmodule

```

Fig. 12



module r\_rcvr(port, rtmc, sak, rak, stall, rst, clk);

input [31:0] port;  
input [19:0] rtmc;  
output sak, rak;  
input stall, rst, clk;  
reg [37:0] in;  
reg [31:0] data;  
reg [1:0] ak, vc;  
reg p, t, valid;  
reg [1:0] buffer[0:31];  
reg [4:0] bufferhead, buffertail;  
wire empty, pop, push;

// Reassemble the flit based on the phase of clk.

always @(posedge clk)  
 in[18:0] <= rtmc;  
always @(negedge clk)  
 in[37:19] <= rtmc;

// Decode the received flit.

always @(negedge clk)  
 {ak, p, t, vc, data} <= in;  
assign valid = !rst & (vc != 3);

// Propagate the remote receiver acknowledge back to the local  
// sender. Filter any noise while reset is in progress.  
assign sak = !rst ? ak : 3;

// Buffer the local acknowledge for the flit just received.

assign push = ~rst & valid;  
always @(posedge clk)  
 if (rst)  
 buffertail <= 0;  
 else if (push) begin  
 buffer[buffertail] <= vc;  
 buffertail <= buffertail + 1;  
 end

// Pull acknowledges from the other end of the buffer as  
// long as they are not being stalled.

assign empty = (bufferhead == buffertail);  
assign pop = ~rst & ~empty & ~stall;  
assign rak = pop ? buffer[bufferhead] : 3;  
always @(posedge clk)  
 if (rst)  
 bufferhead <= 0;  
 else  
 bufferhead <= bufferhead + 1;

raven\_verify\_channel #(33,2)  
 raven(port, {p, data}, vc, t, valid, 1'h0, 1);

endmodule

Fig. 13

#include <raven/diagnostic.h>

namespace mp {

// Port definitions

const int AIN = 0;  
const int AOUT = 1;  
const int BIN = 2;  
const int BOUT = 3;  
const int CIN = 4;  
const int COUT = 5;  
const int DIN = 6;  
const int DOUT = 7;  
const int ASTALL = 8;  
const int BSTALL = 9;  
const int CSTALL = 10;  
const int DSTALL = 11;  
const int MISCIN = 12;  
const int MISCOUT = 13;  
const int RST = 14;

// Command definitions

const int PUT = 0x0;  
const int GET = 0x1;  
const int ERROR = 0xf;

// Register definitions

const int LUT = 0x0000;  
const int MISCIN = 0x1000;  
const int MISCOUT = 0x1004;  
const int LOCK = 0x2000;

// Define a 32 bit register with error flag.

struct flit : public raven::reg<31,0> {  
 raven::reg<0,0> err;  
};

Fig. 14a

// Define a message, include all possible fields.

```
struct msg {
    raven::reg<7,0> dest;
    raven::reg<7,0> src;
    raven::reg<3,0> cmd;
    raven::reg<3,0> mask;
    raven::reg<7,0> id;
    raven::reg<0,0> err;
    flit addr;
    raven::vector<flit> data;
    int vc;
    msg() {}
    msg(const raven::bundle & other);
    operator raven::anybundle();
};
```

// Define port encoding to accept a node and a location.

```
struct port {
    int node, loc;
    port(int p) : node(p >> 8), loc(p & 0xff) {}
    port(int n, int l) : node(n), loc(l) {}
    operator int() { return (node << 8) | (loc & 0xff); }
};
```

// Make all kernel constructs visible.

```
using namespace raven;
```

```
}
```

// Enable diagnostic line numbering.

```
#include <raven/lines.h>
```

CC BY-SA 4.0 licensed presentation

Fig. 146

Fig. 15

```
static int
parity(const raven::reg & x)
{
    int p = 1;
    for (int i = 0; i < 32; i++)
        p ^= x[i];

    return p;
}

mp::msg::msg(const raven::bundle & x)
{
    // Decode head flit.
    dest = x[0](31,24);
    src = x[0](23,16);
    cmd = x[0](15,12);
    mask = x[0](11,8);
    id = x[0](7,0);
    err = x[0](32) ^ parity(x[0](31,0));
    vc = x.vc;

    // Decode address flit for request messages longer than a
    // single flit.
    int w = 1;
    if (vc == 0 && x.size() > 1) {
        addr = x[w](31,0);
        addr.err = x[w](32) ^ parity(x[w](31,0));
        w++;
    }

    // Decode data flits, if any.
    for (int i = w; i < x.size(); i++) {
        data[i - w] = x[i](31,0);
        data[i - w].err = x[i](32) ^ parity(x[i](31,0));
    }
}

mp::msg::operator raven::bundle()
{
    raven::bundle<32,0> x;

    // Encode head flit.
    x[0](31,24) = dest;
    x[0](23,16) = src;
    x[0](15,12) = cmd;
    x[0](11,8) = mask;
    x[0](7,0) = id;
    x[0](32) = err ^ parity(x[0]);
    err = (x[0](32) != parity(x[0](31,0)));
}
```

```
vc = x.vc;
```

```
// Encode address flit for request messages.
```

```
int w = 1;
```

```
if (vc == 0) {
```

```
    x[w](31,0) = addr;
```

```
    x[w](32) = err ^ parity(x[w](31,0));
```

```
    w++;
```

```
}
```

```
// Encode data flits, if any.
```

```
for (int i = 0; i < data.size(); i++) {
```

```
    x[i + w](31,0) = data[i];
```

```
    x[i + w](32) = err ^ parity(x[i](31,0));
```

```
}
```

```
}
```

Fig. 16  
#include <raven/mp.h>

```
int  
main()  
{
```

```
// Write LOCK register.
```

```
mp::msg m;  
m.cmd = PUT;  
m.addr = LOCK;  
m.data[0] = 0xdeadbeef;  
m.vc = 0;  
mp::apply(AIN, m, "Write LOCK request" );
```

```
// Verify response and wait for completion.  
m.data.length(0); // remove data but retain header information  
m.vc = 1;  
mp::verify(AOUT, m, "Write LOCK response" );  
mp::await("Write LOCK register response");
```

```
// Simultaneously read LOCK from all ports  
m.cmd = GET;  
m.vc = 0;  
mp::apply(AIN, m, "Read LOCK request A" );  
mp::apply(BIN, m, "Read LOCK request B" );  
mp::apply(CIN, m, "Read LOCK request C" );  
mp::apply(DIN, m, "Read LOCK request D" );
```

```
// Construct possible outcomes.  
mp::msg yes, no;  
yes.cmd = no.cmd = GET;  
yes.addr = no.addr = LOCK;  
yes.data[0] = 0xdeadbeef;  
no.data[0] = 0;  
int x;
```

```
// Possibility A
```

```
x = mp::verify(AOUT, yes, "Read LOCK response");  
x = mp::verify(BOUT, no, "Read LOCK response", x);  
x = mp::verify(COUT, no, "Read LOCK response", x);  
mp::verify(DOUT, no, "Read LOCK response", x);
```

```
// Possibility B
```

```
x = mp::verify(AOUT, no, "Read LOCK response");  
x = mp::verify(BOUT, yes, "Read LOCK response", x);  
x = mp::verify(COUT, no, "Read LOCK response", x);  
mp::verify(DOUT, no, "Read LOCK response", x);
```

